# Overview

The message services classes are styled after the SILEX framework. They allow a simple way to route messages based upon the AppID field in the AMQP header. The system is built upon layers, which allow programmers to code to differing levels of complexity.

\LWare\Queue\MessageExceptions

php AMQP Driver

\LWare\Queue\MessageQueue

\LWare\Queue\MessageDaemon

\LWare\Queue\ApiProcess

## MessageQueue

A *MessageQueue* is the library that allows you to connect to the amqp library and maintain connections. It will attempt re-connects when required and provides a simpler interface than the standard PHP library

## MessageExceptions

This contains all of the exceptions that can occur in the library. There is no code here that is usable and programmers should avoid using these exceptions in their code.

## ApiProcess

The *ApiProcess* encapsulates each daemon process that will be called. It provides the run interfaces for calling and resolving parameter references. By itself there are only a few interfaces of interest and only in conjunction with *MessageDaemon*.

## MessageDaemon

A daemon class instance binds call routines, queues and processing into one simple class. The class extends the *MessageQueue* class and uses *ApiProcess* to hold the process information.

## Setup

In order to use the services you must instantiate and initialise the service. All the services attempt to perform error handling for you. They provide fairly simple interfaces to setup the services. To use the *MessageDaemon* class you will need to know how to use the *MessageQueue* class.

# MessageQueue

$srv = new MessageQueue();

Instantiate a new message queue object.

$srv->setConnection( ‘amqp://host’ );

Set the connection string for the host. This is split and handled for you in the object rather than having to set multiple values. The values are:

**amqp://username:password@host:port/vhost**

Any of these may be missing and will be filled in by the PHP defaults. (See PHP documentation at <http://php.net/manual/en/amqp.configuration.php> .

## Connection

After *setup*, you can call *getConnection()* to retrieve an \AMQPConnection object. Unless you need the connection object, it is not necessary to create one; they will be created automatically for you. (Only call if you need the connection object. It is handled internally for you.)

$srv->getConnection();

## Channels

After *setup*, you can call *getChannel()* to retrieve an \AMQPChannel object. There is only one channel object per message service object. These are stored and checked internally to ensure they are valid and open. When you call *getChannel* if none are open, it will call *getConnection()* to create a new one. It is not necessary to call both. (Only call if you need the connection object. It is handled internally for you.)

$srv->getChannel();

## Exchange

Exchanges must exist before you can bind queues to them, so creation of an exchange may be needed even if you have no intention of *publishing* a message.

$srv->createExchange(array( ‘name’=>’xname’, ‘parm’ => ‘value’ ) );

The parameters are an array with key/value pairs, and the values may be simple variables or arrays. Each exchange must have a name attached to it. This is mandatory, even if it is blank (‘’).

There is no need to add *declare* as the exchange function will always issue a *declare* when it is created. Options are:

* type => value (Required)
* name=>string (Required)
* flags => *integer value*
* argument => array( string key , mixed value)
* arguments => array( key => value , key=>value…)
* 'declare' => true (not required)

You cannot call *delete* from the *createExchange()* method.

Example:

$exParm = array(

'name' => 'xtest' ,

'type' => \AMQP\_EX\_TYPE\_DIRECT

);

$qd = new MessageQueue();

$qd->setConnectionString( 'amqp://127.0.0.1' )

->createExchange( $exParm )

Once an exchange has been created, you can issue a *getExchange()* to fetch it. This allows you to manipulate the exchange directly or publish messages using it. The *getExchange*

$exchange = $srv->getExchange( ‘name’ );

Name is required and must match the name passed when creating the exchange.

There are utility functions associated with an exchange:

$src->setAppId( ‘string’ );

Application IDs are how messages are routed. The string is used when publish a message.

$srv->publish(‘name’ ,

$message,

$route\_key,

$flags = \AMQP\_NOPARAM,

$attributes = array()

);

This is same as:

$attributes[ ‘timestamp’] = time();

// This is set by calling *setAppId( ‘appid’)*;

$attributes[ ‘appid’ ] = ‘appid’;

$srv->getExchange( ‘name’ )

->publish( $message, $route\_key , $flags , $attributes );

## Queue

Queues are how you receive messages and must be created before they can be used. There are no functions within *MessageQueue* that performs message retrieval.

$srv->createQueue( array( parms) );

The parameters can be:

* **name** => ‘string’  
  same as *setName( string );*
* **declare** => true   
  Same as *declare()*. This is not required and will be called anyway.
* **flags** => *integer*   
  Same as AMQPQueue setFlags values
* **argument** => array( key, value)  
  same as *setArgument( key , value )*
* **arguments** => array( key=>value, key=>value)   
  same as *setArguments( array() )*
* **bind** => array()  
  same as *bind( array() )*

$srv->getQueue()

This will return an object \AMQPQueue that was created by *createQueue( … )*;

## Utility Functions

$srv->setDebug( Boolean );

Set debugging output to on (true) or off (false)

$srv->isDebug();

Return true (debugging is on) or false (debugging is off)

$srv->printDebug( $msg );

If debugging is on, print out *$msg*. This goes to *error\_log* and is prefaces with ‘Debug: ‘.

$srv->isConnected();

Return true or false, depending upon if there is a connection and a channel that is active. It does not check queue connectivity.

$srv->isQueueConnected()

Return true or false, depending upon if there is a connection and a channel that is active and the queue appears to be open. It does not actually test the queue connection as that would require accessing messages.

Also see *Controlling processing.*

## MessageDaemon

Message daemons build upon the *MessageQueue* class and add additional features. All the functions for creating queues and exchanges must be used to create these objects.

The key to daemons are to run routines when there are matches to *AppId* located in the message envelope header. Registered routines are called when one of the following conditions occur:

* A routine is registered to run when it matches an AppID. This is registered by the *match()* function.
* A routine is registered to always run. This is registered by the *always()* function.
* A routine is registered to run if no matches occur. This is registered by the *noMatch()* function.

Additionally, each registered routine can have *before()* and *after()* routines registered to run when the main routine will be run.

The general calling convention is:

$srv = new \LWare\Queue\MessageDaemon();

$srv->match( ‘OnAppId’ ,

function ( \AMQPEnvelope $env , \AMQPQueue $queue )

{

// processing routine

} ,

‘OptionalTagID’

);

The *match* can also be *always* or *noMatch.* The return from the *match()* function will be an \LWare\Queue\ApiProcess object. The first parameter is what it will match against, the second is the routine to process and the third is an optional tag that must be unique if given.

By chaining from the return from *match/always/noMatch*, you can add pre and post processing routines:

$srv->match( ‘OnAppId’ ,

function ( \AMQPEnvelope $env , \AMQPQueue $queue )

{

// processing routine

} ,

‘OptionalTagID’

)

->before(

function ( \AMQPEnvelope $env , \AMQPQueue $queue )

{

// pre-processing routine

}

) ,

->before(

function ( \AMQPEnvelope $env , \AMQPQueue $queue )

{

// pre-processing routine

}

),

->after(

function ( \AMQPEnvelope $env , \AMQPQueue $queue )

{

// pre-processing routine

}

)

;

To add register additional functions, you must code them separately:

$srv->match(…);

$srv->match(…);

$srv->always(…);

Multiple matches may occur (more than one AppId matching). This will run them in order of definition.

The following may be defined as parameters for a function/method:

\AMQPEnvelope The message (envelope) returned from the *get* or *consume* call.

\AMQPQueue The queue object from the *get* or *consume.* Use to *ack* the message.

MessageDaemon The message daemon object. Required to control processing.

The names and order are not relevant as reflection is used to determine the parameters.

$srv->runOne()

Fetch the message and run any routines that match. This will process only *one* message.

$srv->run( $options )

Fetch all the messages and run any matching routines. This calls *consume* so will continue until a halt is indicated. The *$options* are passed to the *consume()* routine and can only be not passed or *AMQP\_NOACK*. (See the terrible documentation.)

## How to pass data between routines

There are several ways to pass data around:

* Use php $GLOBALS to hold values. This is the simplest way and requires little extra code.
* Use the php closure *uses( &$values )*  to hold the values.

There are no built-in mechanisms in the *MessageDaemon* classes to handle this. To ensure that the values are initialised properly you can use the *always* or *before* handling. For example:

// Setup and initialise the daemon storage before we start:

$srv->always( ‘setup’ , function (){ $GLOBALS[‘daemon’] = array(); } );

// add in all the other steps…

$srv->match( ‘appid’ , ... );

// Post cleanup so storage isn’t just sitting around

$srv->always( ‘setup’ , function (){ $GLOBALS[‘daemon’] = array(); } );

This solves the problem of shared storage without having extra mechanisms required.

# Controlling Processing

To enable finer control of processing there are *MessageDaemon* handling mechanisms. In order to use them your routine must include a reference to *MessageDaemon* in the parameters. The routines you can call are:

$d->setProcessingFlag( $flag );

Set the processing flag to *$flag*.

$d->getProcessingFlag( bool $printable\_form );

Fetch the processing flag. If you pass *true* it will return a printable string for debugging purposes.

$d->checkProcessingBits( $flag );

Return true if the bits in *$flag* are set. This allows simple bit testing for control purposes.

$d->clearProcessingBits( $flag );

Clear the processing bits that are set and correspond to *$flag*.

The flag can be an one of the following:

|  |  |  |
| --- | --- | --- |
| Constant | Value | Description |
| HALT\_BEFORE\_LIST | 0x0001 | When a match is found, the *before* registered processing is halted. This does not affect any other matches. |
| HALT\_AFTER\_LIST | 0x0002 | When a match is found, the *after* registered processing is halted. This does not affect any other matches. |
| HALT\_MAIN\_APPID | 0x0004 | When a match is found, don’t run the main registered process. This is only effective when called from the *before* registered processes. |
| HALT\_THIS\_MATCH | 0x0007 | When a match is found, halt any further registered process, including before, main and after. |
| HALT\_AFTER\_MATCH | 0x0008 | After a match has run, stop running any more processes, this includes *noMatches* and *always* registered processes. |
| HALT\_NOW | 0x00ff | This is the same as performing both a *HALT\_THIS\_MATCH* and *HALT\_AFTER\_MATCH*. Everything halts. |
| HALT\_DAEMON | 0x0100 | This only affects daemon processing. After the match runs the daemon will terminate processing. Used only for *consume*. |

# Example:

// The exchange parameters

$exParm = array(

'name' => 'xtest' ,

'type' => \AMQP\_EX\_TYPE\_DIRECT

);

$qd = new MessageDaemon();

/\* Optionally set debugging for testing purposes \*/

$qd->setDebug( false )

->setConnectionString( 'amqp://127.0.0.1' )

->createExchange( $exParm )

->createQueue( array(

'name'=> 'qtest' ,

'bind' => array( 'xtest' , 'ktest') ) )

->appId( $appid ,function(

\AMQPQueue $q ,

\AMQPEnvelope $env,

MessageDaemon $d )

{ // Processing here…

$q->ack( $env->getDeliveryTag() );

}

);

$qd->run(); // endless run